Rapport de projet de programmation impérative

Implémentation d'un algorithme de Page Rank

Evann DREUMONT, Timothée KLEIN

Département Sciences du Numérique - Première année - Groupe E
2023-2024

Ce rapport rend compte du travail que nous avons réalisé lors de l'implémentation en langage Ada de l'algorithme PageRank initialement proposé par Brin & Page. Cet algorithme est à la base des algorithmes de classement de pages web avec pour base leur popularité et est à base matricielle. Deux méthodes ont été envisagées : méthode "pleine" avec des matrices pleines et méthode "creuse" avec des matrices creuses. Finalement, les deux méthodes ayant des avantages et inconvénients, comme la rapidité avec la méthode pleine, mais peu efficiente en termes de mémoire, alors que, à l'inverse, la méthode creuse est plus lente, mais fonctionne sur de grands jeux de données grâce à sa meilleure gestion de la mémoire. La méthode creuse apparaît donc comme une meilleure approche pour aborder le problème réel de recherche de page web.

Table des matières

1 Introduction

L'algorithme PageRank sert à ordonner des pages web selon leur popularité. L'algorithme considère que la popularité d'une page web dépend du nombre d'autres pages web qui la référencent, ce qui peut être modélisé par un graphe que l'on stocke dans une matrice creuse ou pleine. La méthode creuse, contrairement à la méthode pleine, supporte moins le passage à l'échelle, pour contrer cette problématique, nous avons dû changer de type pour représenter les matrices et mettre en place plusieurs méthodes pour optimiser les opérations.

2 Architecture du code

Le code est constitué du programme principal, page_rank.adb servant de points d'entrée à la ligne de commande et de lancer les différents sous-module utilisé afin d'appliquer l'algorithme itératif.

Ainsi, le code est également composé de 3 sous-modules :
- *Matrice* : Permet toutes les opérations classiques sur les matrices (pleines ou creuses) ainsi que la lecture du fichier .net contenant le graphe de référencement des pages web.
- *Export* : Réalise la sauvegarde des sorties de l'algorithme. Nécessite le module matrice.
- *Trifusion* : Réalise le tri-fusion d'une matrice (ligne) pleine. Nécessite le module matrice.

3 Structure de données et algorithmes

3.1 Structures de données

Pour la méthode pleine, la structure adoptée est un tableau de tableaux. Elle permet un accès en temps constants aux éléments. Pour la méthode creuse, nous avons choisi de faire un tableau de listes doublement chaînées : le tableau contient dans chaque case une colonne de la matrice qui est une liste chaînée, cette structure a l'avantage de permettre de stocker uniquement les valeurs non nulles au cout d'un accès plus lent aux éléments.

3.2 Algorithmes

Deux choix principaux ont été réalisés lors de la conception du code (d'autres ont été faits lorsque :
- *Multiplication matricielle* : afin de faciliter le calcul du produit matriciel (et de le rendre viable), nous avons choisi de transposer la matrice de gauche avant de procéder aux calculs. Cela est dû à notre structure en colonnes : il est plus efficace de parcourir les lignes de la matrice de gauche une fois qu'elle a été transposée.
- *Trifusion* : une fois l'algorithme PageRank lui-même terminé, nous devons trier les poids associés à la popularité des nœuds du graphe. Afin de permettre des performances convenables, nous avons alors choisi l'algorithme tri fusion, plutôt efficace avec sa complexité en $O(nlog(n))$.

4 Logique de travail

4.1 Programmes

Initialement, nous avions pensé faire un programme principal "PageRank" qui s'occuperait de faire toutes les opérations lui-même, sauf le trifusion et les calculs matriciels, ceux-ci dans d'autres modules. Aussi, le module trifusion s'aiderait également du module matrice qui contiendrait le(s) type(s) matrice et toutes les procédures s'appliquant dessus. Nous avons ensuite séparé la lecture du graphe et la sauvegarde des fichiers du fichier principal, permettant de simplifier le code ainsi que faciliter l'implémentation des tests unitaires. Finalement, afin d'optimiser la lecture du graphe, nous l'avons intégrée dans le module matrice.

4.2 Makefile

Dans le but de faciliter la compilation, le nettoyage, les tests et l'utilisation du programme en ligne de commande, nous avons écrit un Makefile, permettant, par exemple, de compiler le programme et de le lancer en mode creux sur ***worm.net***, le tout en une seule ligne :

```sh
make make run ARGS="-C networks/worm.net"
```

Évidemment, il reste toujours possible de compiler le programme avec ***gnatmake*** ainsi que de le lancer (mais pas trop fort) avec des arguments en ligne de commande comme définis dans l'énoncer. Le Makefile n'est qu'un wrapper permettant d'utiliser des scripts appelant ces dits programme de manière classiquement pour vous.

Enfin, pour compiler et lancer les tests, il suffira donc de taper en ligne de commande :

```sh
make test run_test
```

4.3 Intégration Continue (CI)

En parallèle du dépôt de l'école, nous avons décidé d'utiliser un dépôt GitHub, cela nous a permis dans un premier temps de faire tourner nos tests automatiquement en s'appuyant sur les GitHub Actions à chaque nouveau commit. Permettant de vérifier que le code ajouté n'impliquait les introductions de nouveau bug. De plus, ces actions sauvegardent à chaque nouveau changement une version du programme compilé afin de pouvoir tester des versions antérieures. Plus récemment, nous avons implémenté un benchmark de notre code, afin qu'à chaque nouveau commit un commentaire soit ajouté au commit avec le temps mis par le programme pour chacun des fichiers fournis. Dans le but d'avoir des résultats cohérents entre eux, et surtout reproductibles, nous avons lancé un worker sur un ordinateur du bâtiment C, permettant d'éviter que l'action soit exécuté dans un container différent à chaque nouvelle exécution.

5 Benchmark

L'ensemble des tests ci-dessous a été réalisé sur l'ordinateur Goldorak de la salle C204 à l'ENSEEIHT. Les temps d'exécution affichée sont les temps moyens après cinq exécutions.

Avant de lancer le benchmark, il a été nécessaire de lancer la commande suivante pour augmenter la taille de la pile d'appels,

```sh
ulimit -s unlimited
```

Enfin, tous les fichiers .net ont été testés vis-a-vis des fuites de mémoire à l'aide de Valgrind, et aucun n'entraine de fuite de mémoire.

5.1 Méthode pleine

5.1.1 sujet.net

```sh
time ./build/page_rank networks/sujet.net

real    0m0,017s
user    0m0,007s
sys     0m0,007s
```

5.1.2 worm.net

```sh
time ./build/page_rank networks/worm.net

real    0m2,736s
user    0m2,736s
sys     0m0,000s
```

5.1.3 brainlinks.net

```sh
./build/page_rank networks/brainlinks.net

real    61m40,470s
user    61m38,765s
sys     0m1,617s
```

5.1.4 linux26.net

Le fichier est trop volumineux pour la stack pour pouvoir le faire tourner avec cette structure de donnée. D'après des calculs élémentaires, il faudrait plus de 300 gigas de stack pour le faire tourner dans l'état actuel.

5.2 Méthode creuse

5.2.1 sujet.net

```sh
time ./build/page_rank -C networks/sujet.net

real    0m0,006s
user    0m0,003s
sys     0m0,000s
```

5.2.2 worm.net

```sh
time ./build/page_rank -C networks/worm.net

real    0m0,048s
user    0m0,041s
sys     0m0,004s
```

5.2.3 brainlinks.net

```sh
time ./build/page_rank -C networks/brainlinks.net

real    0m16,969s
user    0m16,951s
sys     0m0,008s
```

5.2.4 linux26.net

```sh
time ./build/page_rank -C networks/linux26.net

real    2m25,392s
user    2m25,282s
sys     0m0,036s
```

Nous remarquons alors que malgré les temps d'accès plus lent introduit par la structure de donnée creuse, nous observons une exécution plus rapide en mode creux qu'en mode plein. En effet : la plupart du temps, la matrice étant très creuse, nous permettant d'utiliser cette structure de donnée creuse à

son avantage et de laisser de nombreuses opérations inutiles de côtés. C'est cet avantage qui permet un passage à l'échelle et permet de traiter de plus grand graphe.

6 Grille des raffinages

	Evaluation Etudiant (I/P/A/+)
Respect de la syntaxe	+
Verbe à l'infinitif pour les actions complexes	+
Nom ou équivalent pour expressions complexes	+
Tous les Ri sont écrits contre la marge et espacés	+
Les flots de données sont définis	+
Une seule décision ou répétition par raffinage	A
Pas trop d'actions dans un raffinage (moins de 6)	A
Bonne présentation des structures de contrôle	+
Le vocabulaire est précis	+
Le raffinage d'une action décrit complètement cette action	+
Le raffinage d'une action ne décrit que cette action	+
Les flots de données sont cohérents	+
Pas de structure de contrôle déguisée	A
Qualité des actions complexes	+

7 Difficultés rencontrées

7.1 Quelle structure de données pour le mode creux ?

La première difficulté à laquelle nous nous sommes confrontés a été de déterminer quelle structure de donnée utilisée pour le mode creux. Nous avons implémenté plusieurs types de structures avant celui actuel. En effet : la toute première version du code utilisait une liste chainée simple à laquelle on ajoutait linéairement les lignes par lignes. Cette méthode avait l'avantage d'être simple à implémenter, mais le gros désavantage d'être très inefficace dans les calculs et le parcours de celle-ci. Cependant, cette première itération nous a permis d'implémenter notre type Matrice avec un discriminant, nous évitant d'avoir deux modules Matrice distinct, mais seulement un seul, un module prenant en charge les deux types en fonction d'un discriminant.

Dans un second temps, dans le but d'améliorer les performances, nous avons implémenté un nouveau mode creux, cette fois-ci une liste chainée représentant chaque colonne et ayant pour valeur une liste chainée représentant cette fois-ci les lignes. Le désavantage de cette méthode est encore la rapidité, c'est pourquoi nous avons au final retenu le tableau (colonnes) de liste chainée permettant un

accès en temps constant à la colonne et un parcours rapide (dans le cas où la matrice est très creuse) des différentes lignes.

7.2 Comment aller vite ?

Dans ce projet, nous nous sommes centrés sur l'efficacité et la rapidité de notre programme, la plus grosse difficulté a été d'optimiser l'algorithme pour que celui-ci tourne en des temps raisonnables sur les grands graphes. Les modifications ont été faites en plusieurs étapes. Premièrement, nous avons optimisé la lecture des fichiers graphe, ceux-ci étant, pour les gros fichiers, trié dans l'ordre croissant des nœuds, ainsi en gardant en mémoire les derniers curseurs utilisés, nous pouvons insérer de nouvelles arêtes sans perdre de temps à reparcourir la matrice. Cependant, pour les fichiers qui ne seraient pas dans l'ordre et éviter de reparcourir, nous avons introduit un chainage double permettant de retourner en arrière lorsque le point a ajouté est proche du point actuel et de repartir du départ sinon.

Ensuite, nous avons également amélioré notre implémentation de l'algorithme du tri fusion. De plus, nous avons constaté qu'à chaque fois que nous utilisons le tri, c'était sûr des matrices complètement remplies sans 0, alors nous avons déduit qu'il était intéressant d'utiliser des matrices utilisant la structure de donnée pleine pour le tri permettant alors un tri plus rapide.

Enfin, le gros de l'optimisation reste dans l'itération de Page Rank, en effet : nous nous sommes rendu compte qu'il n'était pas nécessaire de calculer à chaque fois la matrice G, mais que nous pouvions tout faire dans le produit. Cependant, cette première solution, a pour effet d'avoir deux boucles imbriquées, et donc une complexité en n^2, ce qu'on a alors cherché à réduire.
Nous avons alors observé que nous pouvions calculer au préalable les termes du produit itératif correspondant aux lignes vides pour pouvoir uniquement les ajouter au nouveau coefficient de π à chaque produit. Cette première implantation reparcourait le tableau contenant le nombre de coefficients de chaque ligne a chaque itération et nous avions une complexité de $2n$. Au final, nous avons introduit un tableau contenant uniquement les liste vide dans le type matrice afin de réduire les nombres d'itérations inutiles dès la seconde itération une fois que celui-ci était peuplé.

Ces différentes optimisations nous ont permis de passer d'un temps d'exécution pour *linux26.net* de près de 45 heures à seulement 2 minutes et 30 secondes, une amélioration très satisfaisante.

8 Conclusions

Finalement, nous avons pu produire un algorithme de Page Rank fournissant des résultats quasi identiques à ceux de l'énoncé et surtout relativement efficace temporellement. Nous avons aussi vu que l'algorithme creux reste, malgré un apriori inverse, plus rapide que l'algorithme plein tout en étant plus efficace en termes de mémoire. En d'autres termes, l'algorithme creux est meilleur que l'algorithme plein.

Ce projet reste tout de même imparfait, nous pourrons citer par exemple l'implémentation matrice beaucoup trop longue qui rend difficile la lecture du code. Ou encore le fait d'avoir à la base voulu implémenter un module totalement générique pour les matrices alors que seulement peu de fonctions sont réellement utile rendant là aussi le code peu lisible. Avec ce projet, nous avons atteint un état ou un refactoring commence à se sentir nécessaire (sinon ça va commencer à nous hanter).

En outre, nous aurions également pu implémenter de la parallélisation pour pouvoirs faire les itérations de l'algorithme encore plus rapidement, mais faute de temps et de documentation (il est plutôt difficile de trouver ce qu'on cherche en Ada sans devoir lire beaucoup de documentions).

9 Apports personnels

Nous arrivons à la fin du projet, et au cours de celui-ci, nous avons pu apprendre à travailler en équipe et a bien organisé notre travail en groupe. Mais il nous a surtout permis de comprendre l'importance d'une bonne structure de donnée et d'une réflexion à l'avance de celle-ci afin d'éviter les différentes fausses routes que nous avons pu emprunter. Mais c'est bien connu que c'est en codant qu'on devient coderons non ? Ce projet nous a permis également de se rendre compte qu'il faut éviter les boucles imbriquées pour permettre un passage à l'échelle sans faire exploser le temps de calcul.

Ce fut une expérience enrichissante avec un sujet intéressant, mais nous avons peut-être manqué de temps pour implémenter toutes les idées d'amélioration de l'algorithme que nous aurions pu avoir.

10 Annexes

10.1 Répartitions des taches

Module	Spécifier	Programmer	Tester	Relire
Matrice	ED	ED	ED	TK
Page Rank	TK	TK	TK	ED
Trifusion	TK	TK	TK	ED
Export	TK	ED	ED	TK